AFRL-IF-RS-TR-2006-242
Final Technical Report
July 2006

DEVELOPMENT OF A SIMULATION MODEL FOR P SYSTEMS WITH ACTIVE MEMBRANES

SUNY Institute of Technology

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

AFRL-IF-RS-TR-2006-242 has been reviewed and is approved for publication

APPROVED: /s/

THOMAS E. RENZ
Project Engineer

FOR THE DIRECTOR: /s/

JAMES A. COLLINS
Deputy Chief, Advanced Computing Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* JULY 2006	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Jan 05 – Jan 06
4. TITLE AND SUBTITLE DEVELOPMENT OF A SIMULATION MODEL FOR P SYSTEMS WITH ACTIVE MEMBRANES		**5a. CONTRACT NUMBER** FA8750-05-2-0042
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER 61101E
6. AUTHOR(S) Digen Das		**5d. PROJECT NUMBER** NBGQ
		5e. TASK NUMBER 10
		5f. WORK UNIT NUMBER 02
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SUNY Institute of Technology, P. O. Box 3050, Utica New York 13504-3050		**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/IFTC, 525 Brooks Rd, Rome New York 13441-4505		**10. SPONSOR/MONITOR'S ACRONYM(S)**
		11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2006-242

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA#06-497

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Membrane Computing (MC) is a branch of Natural Computing, which abstracts from the structure and the functioning of living Cells. The concept was introduced by Gheorghe Paun of the Romanian Academy, Romania, in the late nineties (14, 15). Membrane computing models are commonly known as P (Priority) Systems. These systems perform distributed parallel computing, processing multi-sets of objects synchronously, in compartments delimited by a membrane structure. This report describes a software application, DasPsimulator created in Java. This is a simulation model similar to SimCm (4).While SimCm is limited to its capability to simulate only the P dissolution operation, the DasPsimulator is capable of simulating Membrane Division, Membrane Creation and Membrane String Replication operations. This is a first step to cross the interface between simulation and a Distributed implementation of P Systems able to capture the parallelism existing in the membrane computing area. The tool is user friendly, allowing the user to follow the evolution of a P system in a visual way. The simulator can be used to perform closer inspection of P system theory by the advanced researcher and it can also be helpful to individuals interested in learning and understanding how P systems work.

15. SUBJECT TERMS
Membrane Computing, Natural Computing, Parallel Computing, P Systems

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Thomas E. Renz
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UL	21	**19b. TELEPONE NUMBER (*Include area code)***

Table of Contents

List of Figures

Section 1. Summary

The researchers designed a universal and parallel simulator for a special class of P (Priority) systems known as transitional systems. The DasPsimulator is highly configurable and can in principle be used to evolve membrane systems of any complexity as long as the computing device running the simulation provides sufficient resources. The architecture of the DasPsimulator allows the user to reuse the same module anywhere in the hierarchy of a membrane system and independently of the number of rules and objects to be stored within it. The results have shown that transitional membrane systems can be modeled and simulated very efficiently including membrane dissolution, membrane creation and string replication.

The Model-View Controller (MVC) architecture was used for the development of this interactive system, where the user interfaces are changeable. The DasPsimulator is composed of several different components. In the first one, Model, functional qualities and type abstract data are found. The second component, View, is responsible for showing the results to the user through a graphical interface and the third component, Controller, is in charge of the requests made by the user.

1.1 Subsystems:

Subsystem I contains the simulator engine that contains all of the functional qualities of the basic P systems and type abstract data. This subsystem is formed by combination of the following Java packages: NAryTree (implementing the tree types to represent the membrane structure and the computation tree), List, Membrane, Multi-set, Rules, and Simulator.

Subsystem II includes all the classes related to the Graphical User Interface (GUI) that interacts with the user and is formed by the Java packages: Interface, user Data, Parse Rules, Serialization, and Help.

1.2 Engine of the Simulator

The engine is built upon two fundamental components. The first one is the simulator that includes the algorithms to simulate the processes and computations produced inside a membrane system; it also contains the functional qualities of the system, with the task of starting the initial configuration of the P system and constructing the initial configuration of the associated computation tree. The second component includes all the type abstract data in order to support the membrane structure and its content (multi-sets of objects and rules), and contains the type data necessary for the creation and storage of the applicability multi-sets.

1.3 Graphical User Interface (GUI)

The GUI is the part of the system which allows the user to interact with the application. The Swing Java package was used in the construction of this GUI. The package includes the AWT (Abstract Windows Toolkit). The GUI also uses the JPanel Java class.

The following sections detail all the pertinent aspects of the development of the DasPsimulator, and conclude with future plans for the enhancement of the simulator.

Section 2. Introduction

Membrane Computing (MC) or P systems, initiated by G. Paun in 1998 [14], is a highly parallel, though theoretical, computational model inspired by biochemistry and by some of the basic features of biological membranes. Whereas Paun's membrane computing amalgamates in an elegant way membranes and artificial chemistries, various other systems with membranes also exist. For example, Langton's self-replicating loops [10] make use of a kind of membrane (i.e., a state of the Cellular Automata) that encloses the program. The embryonic projects, on the other hand, use cellular membranes to divide empty space into a multi-cellular organism [12, 20]. Explicit membranes (i.e., membranes with a material consistence) are not always required: the POEtic1project [22], for example, is based on a hierarchical organization of molecules and cells with implicit separations.

The question of whether to simulate systems in software or to implement them in specialized hardware is not a new one (see for example [9]). With the advent of Field Programmable Gate Arrays (FPGA) [21, 24], however, this question took something of a back seat since simulation in Java is relatively straightforward and inexpensive to rapidly build (or rather configure) as compared to FPGAs. P systems are usually implemented and simulated on a standard computer using an existing simulator (such as the SimCM P System simulator [4], or one of the recently proposed distributed software simulators [5, 18]) or a custom simulator. As Paun stated, “it is important to underline the fact that ‘implementing’ a membrane system on an existing electronic computer cannot be a real implementation, it is merely a simulation. As long as we do not have genuinely parallel hardware on which the parallelism of membrane systems could be realized, what we obtain cannot be more than simulations, thus losing the main, good features of membrane systems.” [15, p. 379]. The encouraging part is that today with a distributed system design, high parallelism is now possible.

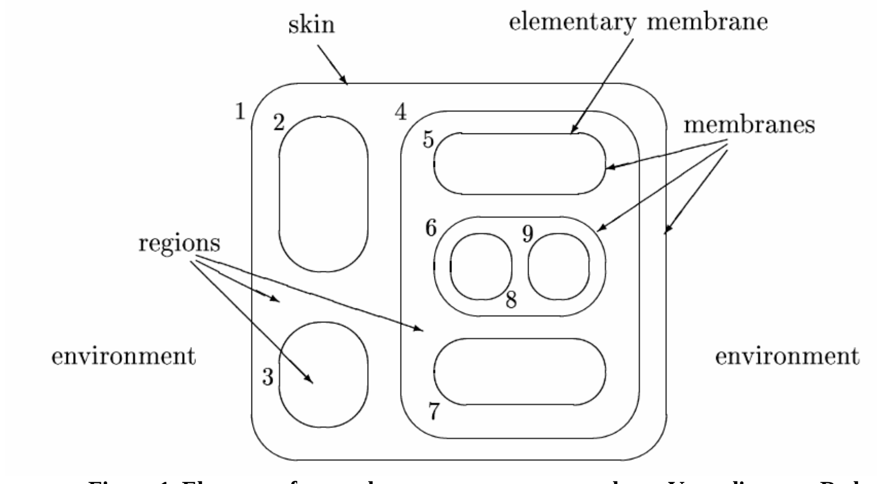


Figure 1. Elements of a membrane system represented as a Venn diagram. Redrawn from [15].

A classical P system (see [15, 16] for a comprehensive introduction) consists of cell-like membranes placed inside a unique “skin” membrane (see Figure1). Multi-sets of objects, usually multi-sets of symbols, objects and a set of evolution rules are then placed inside the regions delimited by the membranes. Each object can be transformed into other objects, can pass through a membrane, or can dissolve or create membranes. The evolution between system configurations is done non-deterministically by applying the rules synchronously in a maximum parallel manner for all objects that are able to evolve. A sequence of transitions is called a computation. A computation halts when a halting configuration is reached, (i.e., when no rule can be applied in any region). A computation is considered successful if and only if it halts.

The following P systems with boundary rules were implemented in the DasPsimulator:

- Communication rules: *xx’ [i y’y → xy’[i x’y]]*
- Evolution rules: [i y → [i y’]] Evolution-Communication P systems
- Communication rules: *(x,in), (y,out), (x,in; y,out)*
- Evolution rules: *y → y’* where *x, x’, y, y’* represent multi-sets of arbitrary size

P systems are not intended to faithfully model the functioning of biological membranes; rather they form a sort of abstract artificial chemistry (AC): "An artificial chemistry is a man-made system which is similar to a real chemical system"[6]. AC's are a very general formulation of abstract systems of objects which follow arbitrary rules of interaction. They basically consist of a set of molecules S, a set of rules R, and a definition of the reactor algorithm A. By abstracting from the complex molecular interactions in Nature, it becomes possible to investigate how the AC's elements change, replicate, maintain themselves, and how new components are created. To be able to efficiently implement P systems in Java code the researchers had to modify classical P systems in the following two aspects:

1. The rules are not applied in a maximum parallel manner but follow a predetermined order.

2. The P system is deterministic, (i.e., for a given initial configuration, the simulation always halts in the same halting configuration).

The primary reason is that a straight-forward implementation of classical P systems would have been too expensive in terms of computer resources required. The researchers were primarily interested in a minimal software implementation and not by a faithful classical P systems implementation. More details on the rules embedded in the DasPsimulator are summarized below:

- Rules are able to perform operations for modifying the membrane structure:

  - membrane creation: [i a]i→ [j b]j
  - membrane division: [i a]i → [k b]k[j c]j
  - membrane duplication: [i a]i → [k b [j c]j]k
  - membrane dissolution: [i a]i → a where a, b are objects and i, j, k are labels of possible membranes

- Communication and Evolution rules assume the form:

[i a → v]i, [i a]i → [i b]i, [i a]i → [i b]i

where a, b are objects and i, j, k are labels of possible membranes

Section 3. Methods, Assumptions and Procedures

In this section, the implementation of the membrane system shall be described in detail. The implementation basically supports P systems with priority using membrane dissolution, creation and string replication. The resulting design is a universal membrane module that can be instantiated and used anywhere in a membrane system.

For the current implementation, the researchers chose Java 2 SDK as the software development platform for designing the P systems simulation. Alternatively, a hardware based P system implementation [17, 21] is an array of (a usually large number of) logic cells placed in a highly configurable infrastructure of connections. Each logical cell can be programmed for a certain function (see also [21] for more details). In addition, once a design has been modeled and simulated via the DasPsimulator it may be subsequently transferred to a full customized Application Specific Integrated Circuit (ASIC) technology, which would in principle provide even better performance.

3.1 Membrane Structure

The membranes described in this implementation are without a material consistence, (i.e. they do not exist as physical manifestations). Therefore, reference to the membrane actually refers to its contents, (i.e., to the multi-sets of objects and the evolution rules of the region it encloses (see Figure 2). The relationships between the membranes, which are represented in this implementation, are illustrated using Venn diagrams as transition containers. A collection is basically only used when objects are being transferred between two membranes.

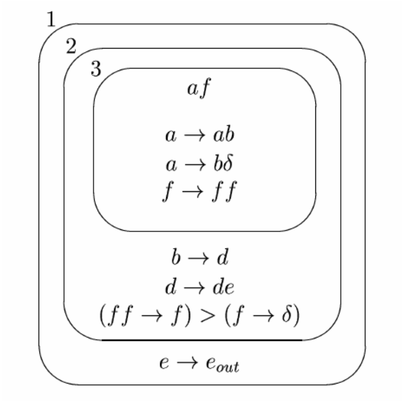


Figure 2. Membranes are not explicitly represented but are defined by its contents.

3.2 Configuring a Membrane System

As an illustrative example, let us assume that we would like to construct the initial membrane system configuration, as depicted in Figure 3 (right), from a single membrane system (i.e., called “mother cell”, left). The following questions immediately arise: (1) how do we represent the description of the membrane system in a compacted form (i.e., the genome), (2) how do we build the membrane system’s initial configuration from this description, and (3) how does the system know when the configuration is terminated and when it can start evolving in “normal” mode? The researchers termed the process of building an initial membrane system configuration a configuration or self-configuration. Thus configuring (or self-configuring) the system happens before the membrane system starts evolving its initial configuration. From a more biological point of view, both steps together are similar to the development of a cell’s structure and functionality.

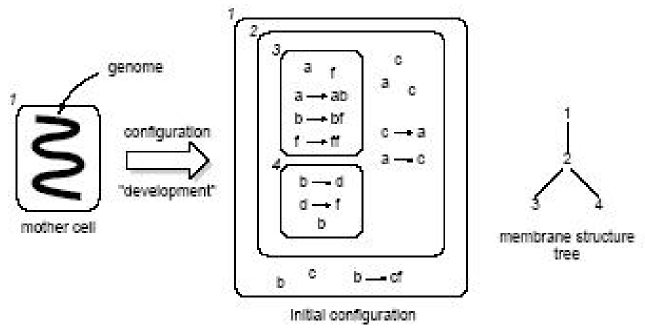


Figure 3. Constructing a membrane system from a single membrane (mother cell), which contains the description of the entire system to be constructed.

As explained for example in [15, p. 301], the membrane creation rule:

cre =a -> $[_i v]_i$,

where a is an object, *v* is a string representing a multiset of objects, and *i* is the membrane label, allows the creation of a new membrane with the label *i* and the contents as given in *v*. For our purposes, the string *v* contains objects and also evolution rules, e.g., v = ab^2c(a -> b)(b -> c). For enhanced readability, the evolution rules shall be put in parenthesis.

The multisets of objects and evolution rules of each membrane compartment contained in Figure 3 are as follows:

v_1 = bc(b -> cf)
v_2 = ac^2(c -> a)(a -> c)
v_3 = af(a -> ab)(b -> bf)(f -> f^2)
v_4 = b(b -> d)(d -> f)

Now, let us assume that the skin membrane [1] has already been created and that it contains a description in some form of the entire membrane system to be built. Here we consider the initial multisets of rules r_1 and objects w_1 that would have to be placed inside this single membrane system in order to obtain the desired initial configuration after some time. A straightforward solution is as follows:

w_1 = M1
r_1 = (M1 -> $v_1[_2 v_2$M3M4(M3 -> $[_3 v_3]_3$)(M4 -> $[_4 v_4]_4$)$]_2$)

The symbols M1, M2, and M3 are used as auxiliary symbols for initiating the membrane creation process in each membrane compartment, v_1, v_2, v_3, v_4 are the multisets of rules and objects as specified above. The only symbol contained in the multiset w_1 (i.e., M1) shall be called seed symbol, the rule r1 genome. The very first step consists in applying rule r_1 to the only object M1 present in the single membrane system. This step creates membrane number two and puts the "building plans" for the next inner cells within its compartment. The process continues until membranes three and four are created. From a mathematical point of view, this can be compared to a recursive construction of the underlying tree structure that represents the membranes. Note that the construction rules remain in the compartments after a successful construction. The reader should note that during the construction process, those membranes whose construction has already completed, will start evolving. Assuming that creating a new membrane can be done within one time step (i.e., one macro-step), the construction time might play a role for deeply nested hierarchies, although one might always take that into consideration while designing the chemistry. Here we propose as an illustrative example another pragmatic solution by introducing a special object D^T, that prevents the chemistry from evolving in the compartment where this object is present as long as $T > 0$. T, which specifies the number of macro-steps the cell will be inactive, is automatically decremented during each macro-step. One can compare this symbol with a counter that decrements T at each time step until it reaches 0. The symbol is automatically removed from the chemistry once T = 0. By expanding the multisets of objects in the genome as following, the membrane system will first be completely built before it starts evolving:

v_1 = bc(b -> cf)D^2
v_2 = ac^2(c -> a)(a -> c)D^1

Since the multisets v_3 and v_4 are at the lowest level in the hierarchy (i.e., the membrane structure's tree leaves), no delay is necessary for them and they can start evolving immediately once created. Membrane 2 (v_2) requires a one-unit delay, the outer membrane compartment v1 two-units.

Note that there are other ways to implement delays and to stop the evolution of the chemistry. In particular, one might convert all rules (e.g., a -> b) into co-operative rules (e.g., *at* -> b) by adding a special symbol (e.g., *t*), which has to be present in the compartment for the rules to be applied. These symbols would then be generated by another special rule.

3.3 Self-Inspection and Self-Replication

Csuhaj-Varj´u et al. [25] proposed a divide-rule denoted as, $[h^{a}]h$ ->$[h^{\prime}b]h^{\prime}$ $[h^{\prime\prime}c]h^{\prime\prime}$, which replicates objects and membranes in *h* alike and puts them into two new membranes $h^{\prime}$ and $h^{\prime\prime}$, at the exception of object a, which is replaced by b in $h^{\prime}$ and by c in $h^{\prime\prime}$. The researchers felt that such a rule is certainly useful from the theoretical point of view, but that (1) it leaves out very important aspects of self-replication, (2) that the rule is too complex compared to other rules to be implemented with a reasonable number of micro-steps, and (3) that it is too complicated to be efficiently implemented in software. The researchers therefore proposed a more pragmatic software-oriented approach in this section, which allows to self-replicate a membrane system (i.e. obtain an identical copy) by means of self-inspection. In the previous section, the configuration process started from a single membrane (i.e., the mother cell) that contained a single seed symbol and a single evolution rule called genome. In this section, we are interested in how we can obtain the genome and the seed symbol from an existing cell by using a sequence of rather simple instructions, which would be implemented by a reasonable number of micro-steps. The situation is illustrated in Figure 4.

The first step of the process will consist of inspecting the current membrane system and in creating a genome-rule in the skin compartment, which will then be used to create a new mother cell in the environment outside the current membrane. In the following steps, the mother cell will develop, i.e., self-configure, until the membrane system's initial configuration is reached (as explained in the previous section). As a result, we will end up with two potentially identical membrane systems. Note that depending on whether the initial membrane system was in a halting configuration and whether the replicated membrane system starts evolving during the configuration process, there might never exist an instant when the two membrane systems are strictly identical. In order to self-replicate a membrane system, the following mechanisms are required: (1) initiate the process of self-inspection; (2) copy the contents (i.e., objects and rules) of a given compartment; (3) incrementally compose the genome; and (4) create the mother cell outside the current skin membrane.

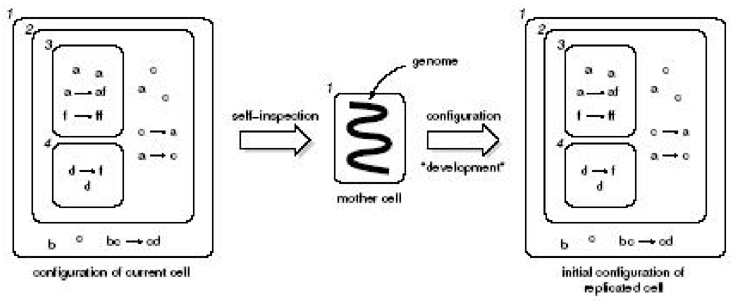


Figure 4. Self-replicating an existing membrane system by self-inspection and sub-sequent configuration.

Let us introduce a new rule, gen = a -> (DC,M*i*, out), which allows the duplication of all objects and evolution rules in the current compartment and which sends the configuration in the form of a special rule to the outer compartment. More specifically, the gen-rule does the following in one macro-step:

(1) duplicates all objects and evolution rules in the current compartment *I* and puts them together in a temporary and virtual multiset DC;
(2) if a rule of the form M*j* -> $[_j \ldots]_j$ is present in the current compartment, an additional symbol M*j*, i.e., the seed symbol, will be added to DC and the rule will be removed from the compartment (but remains in DC);
(3) it sends the rule, M*i* -> $[_i DC]_i$ to the outer compartment; and
(4) if a rule of the form M*i* -> $[_{i.} \ldots]_i$ was already present in the outer compartment, it will simply be replaced by the new rule.

In other words, this rule allows one to incrementally compose the genome from the bottom of the membrane structure to the top by assembling each membrane's configuration on the way in the form of a rule, which will then be used during the construction process as a genome. Here, the symbol M*i* is a symbol, i.e., the seed symbol, not in the symbol alphabet, which is unique for each membrane.

The rule of the form M*i* -> $[_i . . .]_i$ together with the symbol M*i* allows one to create and configure a new membrane i during the construction of the replicated cell. The second step prevents the rule from being used during the self-inspection process. Also, the rule has to be removed because it is not part of the actual configuration and would otherwise be accumulated in case of multiple replication steps. Figure 5 illustrates the application of the gen-rule in a simplified setting. The rule sends the configuration of compartment 2 to the outer compartment. Since no rule of the form M*j* -> $[_j . . .]_j$ is present in the current compartment, no M*j*-symbol is added.

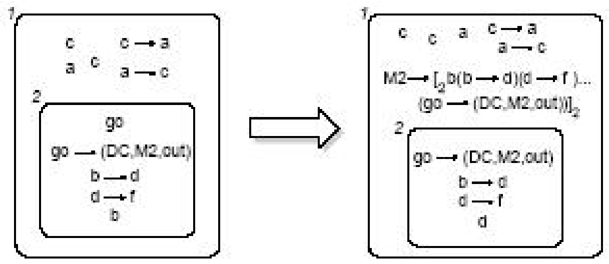


Figure 5. Illustration of the gen-rule.
These rules basically send a copy of the current
objects and evolution rules in the form of a rule to the outer compartment, which
creates a new membrane system in the environment.

If the rule is applied in the top compartment, i.e., the skin compartment 1, the resulting M1 -> $[_1 . . .]_1$ rule, i.e., the complete genome, as well as a seed symbol M1 will be sent to the environment. Once outside the skin membrane, the rule will be immediately applied because of the seed symbol, and a new membrane will be created that contains the genome and the seed symbol for the further construction of the new membrane system (see also Section 3.1). Figure 6 illustrates this procedure, which is not entirely biologically plausible as it involves a moment where the seed symbol and the genome are not surrounded by a membrane, but the researchers felt that this represented an acceptable and more straightforward solution than to introduce another rule, such as a -> $[_1 . . .]_1$, for this particular case.

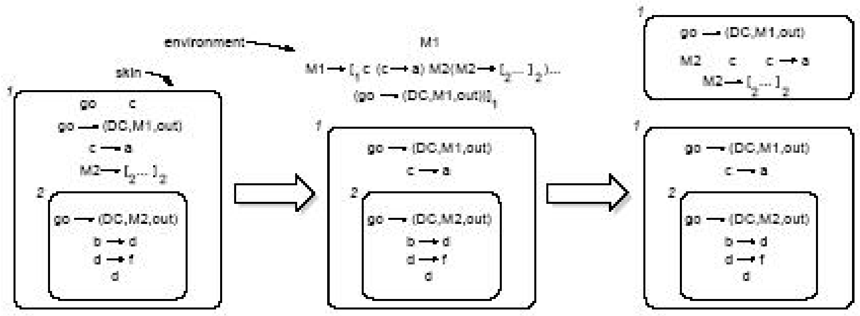


Figure 6. Illustration of the gen-rule when applied in the skin compartment.

In order for the gen-rule to be applied in the correct order (i.e., from the bottom to the top compartment), we need to pre-configure the membrane system with additional symbols and rules. Let us assume that the self-inspection process will be initiated at the bottom of the membrane system, i.e., at the leaves of the underlying tree structure. In Figure 3, the process would thus start in membranes 3 and 4. The entire configuration is then incrementally put together by moving from the leaves to the root of the tree. Here the researchers proposed a self-timed approach, which is illustrated in Figure 7. The idea is that each compartment contains a gen-rule which is being activated by a special symbol (i.e., go). The entire process is initiated by generating a go-symbol in each leaf of the tree, which then triggers the sequential activation of the gen-rules in each compartment. Obviously, if the membrane system changes its own configuration during its normal evolution (i.e., by adding or removing membranes), the gen-rules will have to be modified accordingly, which might not be a trivial undertaking in all cases. For example, if a membrane with label 5 is added in compartment 2 of Figure 7, then the rule would have to be changed from go^2-> . . . to go^3-> . . . and the new membrane would also have to contain a gen-rule. Finally, sending the go-symbol from, for example, the skin compartment to the inner-most membranes, might be realized by means of additional rules not detailed here.

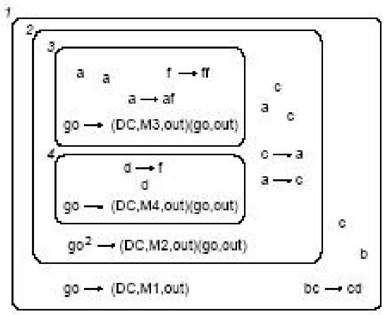


Figure 7. Each compartment has to be pre-configured with a gen-rule in order to allow the membrane system to self-replicate by self-inspection.

The process is initiated by generating a go-symbol in compartment 3 and 4.

3.4. Replicating strings

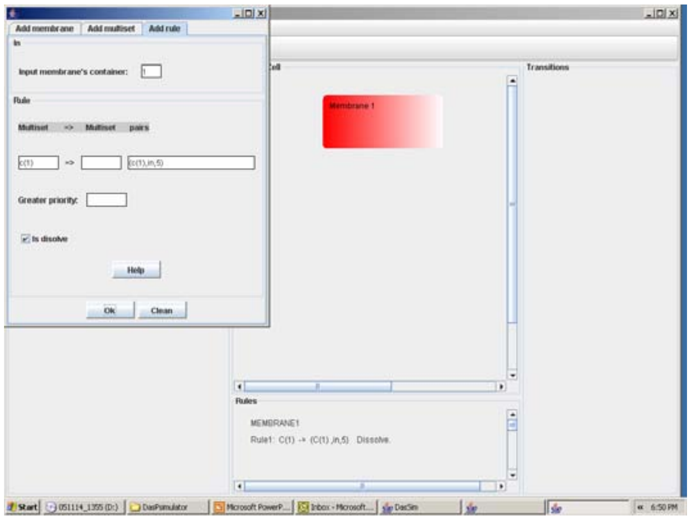
Another powerful way to obtain exponential space sufficient for solving NP-complete problems in polynomial time is to use the replication of string-objects, as considered in P systems with replicated rewriting and in systems with worm objects. It was proven in [26, 27] that the Hamiltonian Path Problem (HPP) and the satisfiability problem (SAT) can be solved in linear time by such systems. (If the replication produces only two new strings, then HPP requires a quadratic time, see [28].) The replication of strings can be obtained not only in a "direct" way, by replicating rules as mentioned above, but also in an "indirect" manner, starting from a conditional way of communicating objects through membranes. The basic idea is to consider certain predicates on strings and communication rules of the form (*II*; in_j); (*II*; out), with the meaning that if *II*(*w*)=true, then the string w must follow the addressing in_j ; out. A variant is to send the string *w* to one of these targets, non-deterministically choosing it, but we may also choose to send the string to all membranes for which a predicate holds true. That is, we replicate the string in as many copies as many communication predicates are true.

Predicates for controlling the string-object communication were considered in [28], but without investigating the computational efficiency of the replication. This was done in [29], for the so-called P systems with valuations, introduced in [30]: a morphism from symbols to integer numbers assigns "valuations" to strings; the sign of this valuation is interpreted as an electrical charge and used for communicating the string as discussed in Section 3.3 (a string of a given polarization goes to a membrane of the opposite polarization, while the neutral strings remain in the same membrane). When a string can go to several adjacent membranes (for instance, it has polarity + and there are several adjacent membranes with polarity −), then the string is replicated and copies of it are sent to all targets. As expected, by using this idea, polynomial solutions of NP-complete problems can be devised; this is illustrated in [30] by SAT and HPP.

3.5 Simulation Design Flow

The membrane system has been programmed using the Java software development kit [3]. The process basically requires three software tools (note that many other tools exist): Eclipse, Java Swing Toolkit and the DasPsimulator.

The Java files are first compiled with Eclipse, which allows the operator to simulate and debug the code on a behavioral level. Once compiled and simulated, the design is synthesized, analyzed, and optimized. This includes all technological relevant details with regards to the chosen schemas, the program flow analysis and the resources required. In addition, Eclipse also outputs the file which is used in the next step by the software. The Eclipse design tool essentially maps (i.e., place and route) the design of the chosen configuration and generates the necessary configuration files. In this case, the Java initialization routine generates the necessary configuration files for simulating the model. A significant part of the code is dependent on the membrane system to be simulated and on its initial values. The DasPsimulator Java application (see Figure 8) allows the user to specify in a convenient way all relevant parameters of the membrane system and then automatically generates the configuration and initialization files as well as several scripts that automate the simulation process.



Input membrane's container: 1

c(1) -> (c(1),in,5)

Greater priority:

[✓ Is disolve]

MEMBRANE1

Rule1: C(1) -> (C(1),in,5) Dissolve.

Figure 8. DasPsimulator Java application interface generates all the necessary configuration files of the membrane system implementation.

Section 4. Results and Discussion

In this section, one example of using the DasPsimulator to solve an NP complete problem is provided and results shall be presented. The DasPsimulator successfully solved the Hamiltonian Path Problem (HPP) in polynomial time and the SAT problem was also solved in polynomial time by applying P system rules with active membranes, by using membrane creation, dissolution and string replication. In the following section the researchers discuss how the DasPsimulator solved the HPP, giving full details, in order to provide the reader with an example of solving an NP complete problem.

4.1 HPP Example

Consider a graph g = (N; E) with the nodes N = $\{a_1; a_2; : : : ; a_n\}$. In order to decide whether a Hamiltonian path exists which starts in a_1 and ends in a_n we construct the P system II with the membrane structure $u = [_0[_1\ ]_1]_0$ (the skin membrane is labeled by 0, and it contains a unique membrane, with label 1), with the object (a_1; 1) present in membrane 1, using the following alphabet of objects:

$V = \{(a_i; j); (a^{/}_i ; j) \mid 1 ,<= i; j <= n\}$ $\{M \quad N \mid M \_= \quad \}$(note that the subsets of N are interpreted as symbol-objects); the possible membranes are labeled by 0; 1; 2; : : : ; n − 1, and the associated sets of rules are as follows:

$R0 = \{N \rightarrow_{yesout}\};$

$Ri = \{(a_i; j) \rightarrow (a^{/}_{k1}; j + 1) : : : (a^{/}_{ksi}; j + 1) \mid (a_i; a_{kr}) \quad E;$ for all

$1 <= r <= si; si >= 1;$ and $1 <= j <= n - 1\}$

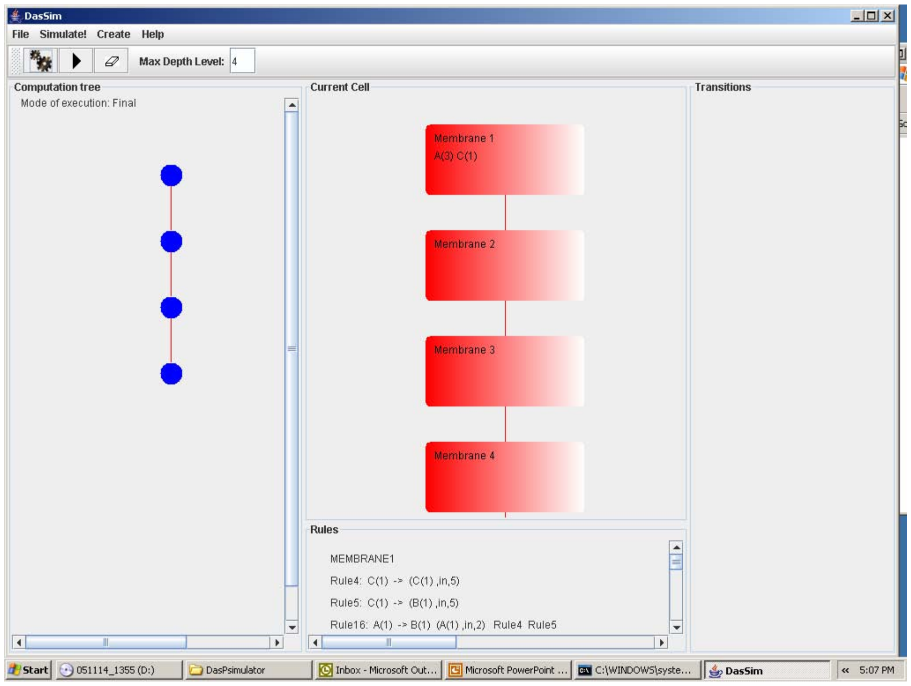
$\{(a^{/}_k; j) \rightarrow [k\ (a_k; j)]_k \mid 1 <= k; j <= n - 1\}$

$\{(a\_n; n) \rightarrow \{an\}\}$

$\{M \rightarrow (M \quad \{a_i\})out \mid M \quad N\}$; for all i = 1; 2; : : : ; n − 1:

The idea behind this construction is the following. The tuple symbols (a_i; j) encode the fact that we have reached node a_i on a path starting in a_1 which has already passed through j nodes. Each object (a_i; j) introduces as many objects of the form ($a^{/}k$; j +1) as many successors of a_i exist in the graph. Then, each object ($a^{/}k$; j + 1) creates a membrane with label k. That is, the paths we create are encoded in the membrane structure (all the paths in the graph g consisting of at most n nodes are “recorded” as paths from the root to the leaf nodes of the tree describing the membrane structure of II). When we reach the node an or the paths already containing n nodes, this process (it takes 2(n−1)−1 steps) is finished, and we pass to the second phase of the computation, that of checking whether or not among the generated paths there is one which is Hamiltonian. This process can start only from object ($a^{/}n$; n), that is, only if we have reached node an after passing through exactly n nodes. After producing an object of the form of a subset of N (at the first step, this is {an}), we exit the membranes, one by one; when we exit membrane i we add the node ai to the current set of nodes. In this way, after at most n steps (one for passing from ($a^{/}_n$; n) to {a_n}, and n − 1 for other nodes), we reach the skin membrane with several objects of the form M   N. Only N can exit the skin membrane, sending out the message yes, that is, we have an output (after 3n−2 steps) if and only if the graph g contains a Hamiltonian path from a_1 to a_n.

The results from this simulation have a special significance in view of the theorem cited above: when we have exponentially many symbol-objects placed in a bounded number of membranes we can simulate the system by a Turing machine of a similar efficiency (with a polynomial slowdown); when one uses an exponential number of string-objects placed in a bounded number of membranes, or an exponential number of objects placed in an exponential number of membranes this is no longer true. We can “explain” these results by the much greater quantity of information stored in a string or in a membrane than in a multiset of symbol objects. The result is depicted in Figure 9. The DasPsimulator successfully solved the HPP problem in polynomial time by using string replication. The algorithm was applied by applying P system rules generating all paths from a specified initial node, and then checking whether or not at least one of these paths is Hamiltonian.



```
MEMBRANE1
Rule4:  C(1)  ->  (C(1) ,in,5)
Rule5:  C(1)  ->  (B(1) ,in,5)
Rule16:  A(1)  -> B(1)  (A(1) ,in,2)   Rule4  Rule5
```

Figure 9. The HPP problem simulated in polynomial time using string replication (ref. page 73 in [15]).

4.2 Model Evaluation

In order to evaluate the performance of this implementation, the researchers simulated multiple membrane models of varying sizes and complexities. The researchers found the resources used are nearly directly proportional to the maximum number of objects. Furthermore, adding the possibility of membrane creation adds complexity and therefore results in a design that is almost twice as large and runs at a much slower speed.

This DasPsimulator has been tested by means of examples from [15] with the following features: membrane dissolution, membrane creation, and string replication consisting of transferring objects to upper- and lower-immediate membrane systems.

Section 5. Conclusions

The researchers presented a universal and massively parallel implementation of a special class of P systems. The architecture of the universal membrane allows us to use the same module anywhere in the hierarchy of a membrane system and independently of the number of rules and objects to be stored within it. The conclusions summarized below were modeled and simulated efficiently using the DasPsimulator:

- Membrane Computing provides computational models that abstract from the living cells' structure and function

- Such models have been proven to be computationally powerful (equivalent to a Turing machine) and efficient (solving NP-Complete problems)

- Membrane Computing defines an abstract framework for reasoning about
  - distribute architectures
  - communication
  - parallel information processing

- Such features are relevant both for Computer Science (Distributed Computing Models, Multi-Agent Systems) and Networking (Modeling and Simulation of Biological Networks)

Section 6. Recommendations

Future work will concentrate on the development and improvement (in terms of speed and resources used) of the existing design. In addition, it is planned to extend the existing design in order to be able to reuse dissolved membranes and in order to apply rules in a fully parallel and nondeterministic manner. In addition, dealing with a larger number of objects would probably require extensive processor capacity. This was not a serious limitation in the current implementation. Furthermore, the researchers also envisage extending the current design to other important classes of P systems such as for example systems with symport/antiport [15, p. 130] and systems with membrane division [15, p. 273] such as those summarized below:

Examples of specialized P system models:

- Energy-Controlled P systems
- P systems with promoters/inhibitors
- P systems with carriers
- P systems with mobile membranes
- Tissue P systems
- Probabilistic P systems
- P systems with elementary graph productions
- Parallel Rewriting P systems

Ongoing research consists of developing a hardware based architecture for membrane systems using FPGA’s, amalgamated with alternative computational and organizational metaphors. Amorphous Membrane Blending (AMB) [19], for example, is an original and not less unconventional attempt to combine some of the interesting and powerful traits of Amorphous Computing [2], Membrane Computing [15], Artificial Chemistries [6], and Blending [7]. One of the goals is to obtain a novel, minimalist, and computational architecture with organizational principles inspired by biology and cognitive science.

References

1. International technology roadmap for semiconductors. Semiconductor Industry Association, retrived from http://public.itrs.net/Files/2001ITRS, 2001.
2. H. Abelson, D. Allen, D. Coore, C. Hanson, E. Rauch, G. J. Sussman, and R.Weiss. Amorphous computing. Communications of the ACM, 43(5):74-82, May 2000.
3. P. J. Ashenden. The Designer's Guide to VHDL. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.
4. G. Ciobanu and D. Paraschiv. Membrane software. A P system simulator. Fundamental Informaticae, 49(13):61-66, 2002.
5. G. Ciobanu and G. Wenyuan. A parallel implementation of the transition P systems. In A. Alhazov, C. Martin-Vide, and G. Paun, editors, Proceedings of the MolCoNet Workshop on Membrane Computing (WMC2003), volume 28/03, page 169, Tarragona (Spain), 2003. Rovira I Virgili University, Research Group on Mathematical Linguistics.
6. P. Dittrich, J. Ziegler, and W. Banzhaf. Artificial chemistries: a review. Artificial Life, 7(3):225-275, 2001.
7. G. Fauconnier and M. Turner. The Way We Think: Conceptual Blending and the Mind's Hidden Complexities. Basic Books, 2002.
8. S. De Franceschi and L. Kouwenhoven. Electronics and the single atom. Nature, 417:701-702, June 13 2002.
9. L. Garber and D. Sims. In pursuit of hardware-software codesign. IEEE Computer, 31(6):12{14, June 1998.
10. C. G. Langton. Self-reproduction in cellular automata. Physica D, 10:135-144, 1984.
11. M. Madhu, V. S. Murty, and K. Krithivasan. A hardware realization of P systems with carriers. Poster presentation at the Eight International Conference on DNA based Computers, Hokkaido University, Sapporo Campus, Japan, June 10-13 2002.
12. D. Mange, M. Sipper, A. Stauer, and G. Tempesti. Toward robust integrated circuits: The embryonics approach. Proceedings of the IEEE, 88(4):516-540, April 2000.
13. N. Mathur. Beyond the silicon roadmap. Nature, 419(6907):573-575, October 10 2002.
14. G. Paun. Computing with membranes. Journal of Computer and System Sciences, 61(1):108-143, 2000. First published in a TUCS Research Report, No 208, November 1998, http://www.tucs.fi.
15. G. Paun. Membrane Computing. Springer-Verlag, Berlin, Heidelberg, Germany, 2002.
16. G. Paun and G. Rozenberg. A guide to membrane computing. Journal of Theoretical Computer Science, 287(1):73-100, 2002.

17. E. Sanchez. An introduction to digital systems. In D. Mange and M. Tomassini, editors, Bio-Inspired Computing Machines: Towards Novel Computational Architectures, chapter 2, pages 13-47. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.
18. A. Syropoulos, E. G. Mamatas, P. C. Allilomes, and K. T. Sotiriades. A distributed simulation of P systems. In A. Alhazov, C. Martin-Vide, and G. Paun, editors, Proceedings of the MolCoNet Workshop on Membrane Computing (WMC2003), volume 28/03, pages 455-460, Tarragona (Spain), 2003. Rovira i Virgili University, Research Group on Mathematical Linguistics.
19. C. Teuscher. Amorphous Membrane Blending and Other Unconventional Computing Paradigms. PhD thesis, Swiss Federal Institute of Technology (EPFL), Lausanne, Switerland, 2004. To be published.
20. C. Teuscher, D. Mange, A. Stauer, and G. Tempesti. Bio-inspired computing tissues: Towards machines that evolve, grow, and learn. BioSystems, 68(2)(3):235-244, February-March 2003.
21. S. M. Trimberger. Field-Programmable Gate Array Technology. Kluwer Academic Publishers, Boston, 1994.
22. A. Tyrrell, E. Sanchez, D. Floreano, G. Tempesti, D. Mange, J.-M. Moreno, J. Rosenberg, and Alessandro E. P. Villa. Poetic tissue: An integrated architecture for bio-inspired hardware. In A. M. Tyrrell, P. C. Haddow, and J. Torresen, editors, Evolvable Systems: From Biology to Hardware. Proceedings of the 5th International Conference (ICES2003), volume 2606 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, 2003.
23. F. Varela, H. Maturana, and R. Uribe. Autopoiesis: The organization of living systems, its characterization and a model. BioSystems, 5:187-196, 1974.
24. J. Villasenor and W. H. Mangione-Smith. Configurable computing. Scientific American, 276(6):54-59, June 1997.
25. Csuhaj-Varju, E., Nola, A.D., Paun, G. Perez-Jiminez, M.J., Vaszil, G., 2005. Editing configurations of P Systems, submitted.
26. J. Castellanos, A. Rodriguez-Paton, G. Paun, Computing with membranes: P systems with worm-objects, IEEE 7th International Conference on String Processing and Information Retrieval, SPIRE, La Coruna, Spain, 2000, pp. 64-74
27. S.N. Krishna, R. Rama, P systems with replicated rewriting, J. Automata, Languages, Combin. 6 (2001) pp. 345-350.
28. P. Bottoni, A. Labella, C. Martin-Vide, G. Paun, Rewriting P systems with conditional communications, in: W. Brauer, H. Ehrig, I. Karhumaki, A Salomaa (Eds.), Formal and Natural Computing. Essays Dedicated to Gregorz Rozenberg, Lecture Notes in Computer Science 2300. Springer, Berlin. 2002, pp. 325-353.
29. C. Martin-Vide, V. Mitrana, P systems with valuations, in I. Antoniou, C.S. Calude, M.J. Dinneen (Eds.), Unconventional Models of Computation, Springer, London, 2000. pp. 154-166.
30. C. Martin-Vide, V. Mitrana, G. Paun, On the power of P systems with valuations, Computacion Sistemas 5 (2001) pp. 120-127.